编译原理
课程设计实验报告

学　　号 1552238

姓　　名 张宏伟

专　　业 计算机科学与技术

授课老师 丁志军

一、功能描述

1. 设计目的

- 掌握使用高级程序语言实现一个一遍完成的、简单语言的编译器的方法；
- 掌握词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现方法；
- 掌握将生成代码写入文件的技术。

2. 设计要求

- 使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。
- 要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用;
- 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。
- 要求输入类 C 语言源程序，输出中间代码表示的程序;
- 要求输入类 C 语言源程序，输出目标代码(可汇编执行)的程序。
- 实现过程、函数调用的代码编译

3. 完成结果

最终完成了一个类 C 语言的编译器，读入 C 语言源程序，最终可以得到可汇编执行的一个可视化演示程序。

二、程序结构

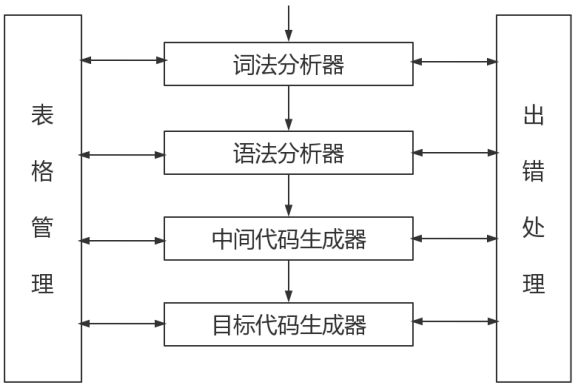


图 2-1 设计目标

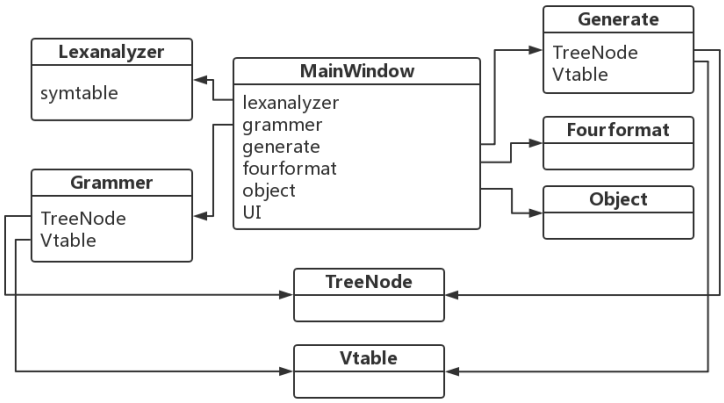


图 2-2 程序总体框架

三、词法分析

1. 操作步骤：

将字符逐个读入，使用数组模拟自动机，以判断属于哪种类型。

a) ‘ ’，\t，\r，直接跳过

b) \n，记录下来，用于后面的行数计算，记为 NL

c) 数字，扫描至数字的结束符(+ = >=等)，记为 NUM

d) 操作符，+ -记为 OP1， * /记为 OP2， ，记为 SEP，;记为 DEL，(记为 LP，)记为 RP，{记为 LB，}记为 RB，> < = >= <= == !=记为 RELOP（在语法和词法的角度，暂且忽略各个符号的真实含义，方便化简）

e) 保留字(int void while if else return)，按照不同的保留字分别记为 INT VOID WHILE IF ELSE RETURN

f) 标志符(自定义的变量名称、main)，记为 ID

g) 注释信息，记为 COM，不在最终的文件里显示

2. 设计思路：

将编译的过程分层，上一级的输出作为下一级的输入。因为每一层其实是翻译的过程，所以每一级可能会伴随信息的损失。如何选取每一集信息的保留，就涉及到每层的功能的要求。

词法分析的功能：

- 断词就是分析输入的一段词的属性。
- 建立符号表，保留每个词的名字以方便后面语义分析
- 去掉空白、注释。

- 保留行号信息，这个有些像加入了冗余，但为了报错的可视化（提示出错行号），所以最终做了保留。

词法分析的过程：

词法分析的过程其实就是一个模拟自动机运行的过程。并根据输入的情况决定下一步跳转到哪一状态或直接输出结果。

1. 每次开始分析的过程是，首先读入一个字符，根据符号的类型选择进入调用不同的函数作为自动机的入口。

   每个函数会返回一个值，0 表示成功识别，正数表示出错行数。

```
bool isOperator(char ch);                    //是否为一元运算符
int comment(char ch, int i);                 //判断是否是注释
int number(char ch, int i);                  //判断是否是数字
int isResw(char *s);                         //判断是否是保留字
int identifier(char ch, int i);              //判断是否是标识符
```

2. 在函数内部不断读取字母，直到遇到不符合的字母，会返回结果，并让读取字母的指针回退（因为多读一个字母）。

   模拟自动机的过程，首先使用一个 int 数组，对应上面的五个函数的遍历过程：

```
int state[5] = { 0 }; //不同记号类型的状态
```

   比如识别操作符的过程，如果读取到一个字符，就进入判断的对应函数

```
else if (isOperator(ch))
{
    states = myOperator(ch, 2); //处理操作符
    clearState();
}
```

```
switch (state[i]) {
case 0: //非以上符号时
    beginp[i] = ftell(source) - 1;
    switch (ch) { ... }
    break;

case 2:
    if (ch == '=') { ... }
    else { ... }
    break;

case 4:
    if (ch == '=') { ... }
    else { ... }
    break;
```

```
case 6:
    if (ch == '=') { ... }
    else { ... }
    break;

case 8:
    if (ch == '=') { ... }
    break;
default:
    return lineNum;
}
```

根据当前所处状态以及输入的情况，作为下一步跳转到另一状态或是跳出此次测试的依据。跳出时在前后两指针之间的内容，也做为下一步的输入。

3. 调用不同的输出函数，并把结果输出在文件中

```
int strPrintf(long begin, long end, tokenType t);        //输出一个字符串单元
void unaryPrintf(char s, tokenType t);                   //输出一个一字母的运算符单元
```

4. 然后调用初始化函数，开始新的词法分析。直到读到了文件位

```
void clearState();                                       //读取头的初始化
```

比较有特色的，就是我们加入了 NL（换行符）这个看似有些冗余的 Token 单元。这样每一个换行可以作为一个单元输出，而不需要把他作为 token 单元的一部分，这样不需要每一个都输出每个单元的行号。只需要数到当前所读到的 NL 单元数，然后作为输出。

使用的输出函数是这个：

```
void addNewLine();                                       //输出换行符
```

四、语法分析

1. 化简语法，使其符合 LL1 规则

- Program ::= <类型> < ID>’(‘ ‘)’<语句块>
- <类型>::=int | void
- <语句块> ::= ‘{‘<内部声明> <语句串>’}’
- <内部声明> ::= 空 | <内部变量声明><内部声明>
- <内部变量声明>::=int <ID><next> ;
- <next> ::= ,<ID><next> | 空
- <语句串> ::= <语句> <语句串> | 空
- <语句> ::= <if 语句> |< while 语句> | <return 语句>; | <赋值语句> ;
- <赋值语句> ::= <ID> =<表达式>
- <return 语句> ::= return retBlock

- retBlock ::= 空 | 表达式
- <while 语句> ::= while ‘(‘<表达式> ‘)’ <语句块>
- <if 语句> ::= if ‘(‘<表达式>’)’ <语句块> elseBlock
- elseBlock ::= else <语句块> | 空
- <表达式>::=<加法表达式> <comp>
- <comp> ::= relop <加法表达式><comp> | 空
- <加法表达式> ::= <项> <op1>
- <op1> ::= +<项><op1> | 空
- <项> ::= <因子> <op2>
- <op2> ::= * <因子> <op2> | 空
- <因子> ::=num | ‘(‘<表达式>’)’ |<ID>

2. 用定义好的枚举类型表示产生式

```
typedef enum { INT, VOID, ID, LP, RP, LB, RB, WHILE, IF, ELSE,
RETURN, ASSIGN, OP1, OP2, RELOP, DEL, SEP, NUM, NL, PROGRAM, TYPE,
SENBLOCK, INNERDEF, INNERVARIDEF, NEXT, SENSEQ, SENTENCE,
ASSIGNMENT, RETURNSEN, RETBLOCK,  WHILESEN, IFSEN, ELSEBOCLK,
EXPRESSION, COMP, PLUSEX, OPPLUSDEC, TERM, OPMULDIV, FACTOR,
EPSILON, END, ERROR}tokenType;
```

3. 符号表

符号表	int	void	ID	(	)	{	}	while	if
Enum	INT	VOID	ID	LP	RP	LB	RB	WHILE	IF

符号表	else	return	=	+ -	* /	> < >= <= == !=	;	,
Enum	ELSE	RETURN	ASSIGN	OP1	OP2	RELOP	DEL	SEP

符号表	num	\n	<Program>	<类型>	<语句块>	<内部声明>
Enum	NUM	NL	PROGRAM	TYPE	SENBLOCK	INNERDEF

符号表	<内部变量声明>	<next>	<语句串>	<语句>	<赋值语句>
Enum	INNERVARIDEF	NEXT	SENSEQ	SENTENCE	ASSIGNMENT

符号表	<return 语句>	<retBlock>	<while 语句>	<if 语句>	<elseBlock>

Enum	RETURNSEN	RETBLOCK	WHILESEN	IFSEN	ELSEBLOCK

符号表	<表达式>	<comp>	<加法表达式>	<op1>	<项>
Enum	EXPRESSION	COMP	PLUSEX	OPPLUSDEC	TERM

符号表	<op2>	<因子>	空	#
Enum	OPMULDIV	FACTOR	EPSILON	END

- PROGRAM -> TYPE ID LP RP SENBLOCK
- TYPE -> INT |VOID
- SENBLOCK -> LB INNERDEF SENSEQ RB
- INNERDEF -> EPSILON |INNERVARIDEF INNERDEF
- INNERVARIDEF -> INT ID NEXT DEL
- NEXT -> SEP ID NEXT |EPSILON
- SENSEQ -> SENTENCE SENSEQ |EPSILON
- SENTENCE -> IFSEN |WHILESEN |RETURNSEN DEL |ASSIGNMENT DEL
- ASSIGNMENT -> ID ASSIGN EXPRESSION
- RETURNSEN -> RETURN RETBLOCK
- RETBLOCK -> EXPRESSION |EPSILON
- WHILESEN -> WHILE LP EXPRESSION RP SENBLOCK
- IFSEN -> IF LP EXPRESSION RP SENBLOCK ELSEBOCLK
- ELSEBOCLK -> ELSE SENBLOCK |EPSILON
- EXPRESSION -> PLUSEX COMP
- COMP -> RELOP PLUSEX COMP |EPSILON
- PLUSEX -> TERM OPPLUSDEC
- OPPLUSDEC -> OP1 TERM OPPLUSDEC |EPSILON
- TERM -> FACTOR OPMULDIV
- OPMULDIV -> OP2 FACTOR OPMULDIV |EPSILON
- FACTOR -> NUM |LP EXPRESSION RP |ID

4. 表达生成式

```
class sentence //用来表示右部生成式
{
public:
    sentence* next; //指向下一个生成式
    tokenType tokens[6]; //生成式集
    int tokenLen; //生成式长度
```

```
};

class generator //用来表示左部的非终结符，相当于 LL1 表中的行名
{
public:
    tokenType head; //左部非终结符名
    sentence* descri; //指向第一个生成式

};
```

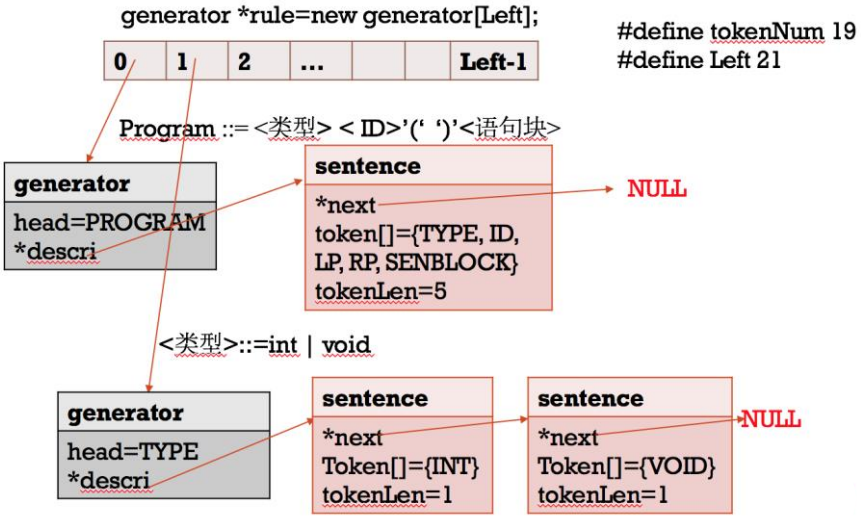


图 4-1

5. 建立 FIRST 集和 FOLLOW 集

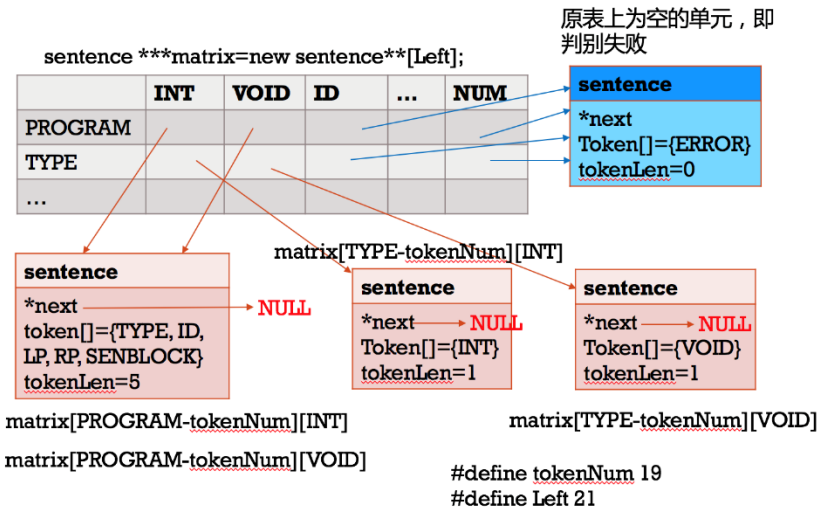
		frist	follow
1		frist	follow
2	program	int-void	#
3	类型	int-void	ID
4	语句块	{	else-E-#
5	内部声明	E-int	if-while-return-ID-E-}
6	语句串	if-while-return-ID-E	}
7	内部变量声明	int	if-while-return-ID-E-int
8	语句	if-while-return-ID	if-while-return-ID-}
9	if语句	if	if-while-return-ID-}
10	while语句	while	if-while-return-ID-}
11	return语句	return	if-while--ID-;
12	赋值语句	ID	if-while-return-ID-;
13	表达式	num-ID-(	)-;
14	加法表达式	num-ID-(	)-;
15	项	num-ID-(	)-;
16	因子	num-ID-(	)-;
17	next	,-E	;
18	comp	relop-E	)-;
19	retBlock	num-ID-(-E	;
20	op1	E-+	)-;
21	op2	E-*	)-;
22	elseBlock	else-E	if-while-return-ID-}

图 4-2 FIRST 集和 FOLLOW 集

非终结符=: 列标：对应的生成式右部 | 列标:对应的生成式右部

- PROGRAM=: INT:TYPE ID LP RP SENBLOCK | VOID:TYPE ID LP RP SENBLOCK
- TYPE=: INT:INT | VOID:VOID
- SENBLOCK=: LB:LB INNERDEF SENSEQ RB
- INNERDEF=: INT:INNERVARIDEF INNERDEF | ID:EPSILON | RB:EPSILON | IF:EPSILON | ELSE:EPSILON | RETURN:EPSILON
- INNERVARIDEF=: INT:INT ID NEXT DEL
- NEXT=: DEL:EPSILON | SEP:SEP ID NEXT
- SENSEQ=: ID:SENTENCE SENSEQ | RB:EPSILON | WHILE:SENTENCE SENSEQ | IF:SENTENCE SENSEQ | RETURN:SENTENCE SENSEQ
- SENTENCE=: ID:ASSIGNMENT DEL | WHILE:WHILESEN | IF:IFSEN | RETURN:RETURNSEN DEL
- ASSIGNMENT=: ID:ID ASSIGN EXPRESSION
- RETURNSEN=: RETURN:RETURN RETBLOCK
- RETBLOCK=: ID:EXPRESSION | LP:EXPRESSION | DEL:EPSILON | NUM:EXPRESSION
- WHILESEN=: WHILE:WHILE LP EXPRESSION RP SENBLOCK
- IFSEN=: IF:IF LP EXPRESSION RP SENBLOCK ELSEBOCLK
- ELSEBOCLK=: ID:EPSILON | RB:EPSILON | WHILE:EPSILON | IF:EPSILON | ELSE:ELSE SENBLOCK | RETURN:EPSILON
- EXPRESSION=: ID:PLUSEX COMP | LP:PLUSEX COMP | NUM:PLUSEX COMP
- COMP=: RP:EPSILON | RELOP:RELOP PLUSEX COMP | DEL:EPSILON
- PLUSEX=: ID:TERM OPPLUSDEC | LP:TERM OPPLUSDEC | NUM:TERM OPPLUSDEC
- OPPLUSDEC=: RP:EPSILON | RETURN:EPSILON | OP1:OP1 TERM OPPLUSDEC | RELOP:EPSILON | DEL:EPSILON
- TERM=: ID:FACTOR OPMULDIV | LP:FACTOR OPMULDIV | NUM:FACTOR OPMULDIV
- OPMULDIV=: RP:EPSILON | RETURN:EPSILON | OP1:EPSILON | OP2:OP2 FACTOR OPMULDIV | RELOP:EPSILON | DEL:EPSILON
- FACTOR=: ID:ID | LP:LP EXPRESSION RP | NUM:NUM

6. LL1 分析表



7. 语法树

语法分析树的输出算法，其实就是树的输出，首先计算每一个要输出节点层数，并且记录每层所有的节点个数，如果为零，说明不需要输出这一层的树枝的表示。否则需要输出“|---”

```
int grammer::analyze(string filename)
{
vector<int> levelCount = vector<int>(128, 0);    / / 当前
vector<int> level = vector<int>(1024, -1);
    ifstream fin(filename, std::ios::in);
    if (!fin){
        return -1;
    }
ofstream fout(OFileName, std::ios::out);
if (!fout) {
    return -1;
}
    string line;
    int lineNum=1;

int point=0;//始终指向栈顶元素
    stack[point]=END; //将终结符压入栈底
    stack[++point]=PROGRAM; //将生成式的开始符号压入栈
```

```
level[point] = 0;
    bool flag=true;
    getline(fin, line);//读取词法分析后的一行数据 显示形式为< ID , -1 >
    string v;
int attribute;
    SplitString(line, v, attribute);//用空格分割行，取出需要的标识符
    int count = getToken(v);
    while(flag){
        while(count==NL){ //忽略换行符
            lineNum++;
            getline(fin, line);
            SplitString(line, v, attribute);
            count = getToken(v);
        }
        if(stack[point] < tokenNum){//栈顶为终结符时
            if(stack[point]==count){
            //栈顶 x 与输入符 a 匹配，将 x 弹出，指针后移，读下一符号，分析。
            for (int i = 1; i < level[point]; i++)
                if (levelCount[i] != 0)
                    fout << special[0];
                else
                    fout << special[2];
            fout << special[1];
            levelCount[level[point]]--;
            fout << strTokens[stack[point]];
            switch (stack[point]) //输出树形结构
            {
            case ID:
            case NUM:
            case OP1:
            case OP2:
            case RELOP:
                fout << " - " << attribute << ' ';
                break;
            default:
                break;
            }
            fout << endl;
               if(getline(fin, line)){
```

```
        //读下一字符，若完成，返回-1，证明语法正确，检验成功
            SplitString(line, v, attribute);
            count = getToken(v);
            point--;
            continue;
        }
        else
            return 0; //正确判断的输出结果为0
    }
    else{
        flag=false;
        //栈顶与当前字符不匹配，产生语法错误，识别失败，跳出循环
    }
}
else if(stack[point]==END){

    if(stack[point]==count){
     levelCount[level[point]]--;
     point--;
        continue;
    }
    else{
        flag=false;//栈顶与当前字符不匹配，语法错误，识别失败，
    }
}
else if(stack[point]>=PROGRAM && stack[point]<=FACTOR){
//栈顶为非终结符
for (int i = 1; i < level[point]; i++)
    if (levelCount[i] != 0)
        fout << special[0];
    else
        fout << special[2];
if(stack[point] != PROGRAM)
    fout << special[1];

fout << strTokens[stack[point]];
switch (stack[point])
{
case ID:
```

```
        case NUM:
        case OP1:
        case OP2:
        case RELOP:
            fout << " - " << attribute << ' ';
        default:
            break;
        }

        sentence *s=new sentence;
           s=matrix[stack[point]-tokenNum][count];//查 LL1 表
           if(s->tokens[0]==ERROR){
           //为空，则发现语法错误，调用出错处理程序进行处理
               flag=false;
           }
           else if (s->tokens[0]==EPSILON){
           //A→ε ，则只将 A 自栈顶弹出。
            fout << " - E";
            levelCount[level[point]]--;
               point--;
           }
           else{//将生成式逆序逐一压入栈中 生成树形结构
            int j = level[point];
            levelCount[j]--;
               for(int i=s->tokenLen-1;i>=0;i--){
                   stack[point] = s->tokens[i];
                level[point] = j + 1;
                levelCount[j + 1]++;
                   point++;
               }
               point--;
           }
        fout << endl;
       }
    }
    return lineNum;//出错，出错行的行号。
}
```

五、中间代码生成

1. 抽象语法树

此阶段的语法树由语法分析阶段产生，是语法分析调整后的结果。表达式采用逆波兰式的方式组织，使操作简便。

2. 三地址码

根据抽象语法树生成三地址码的过程需要维护变量的作用域信息，符号表 gtable 以及跳转指令的标号管理 Qtable。
采用递归的方式表示每一个语句块。

3. 四元式

将三地址码生成四元式，更符合目标代码的结构。

```
int Generate::Recursive(treeNode *p)
{
    //qDebug()<<strTokens[p->type]<<"   "<<p->value<<"   "<<p->lineNum;
    int flag=1;
    //进入语句块
   // gtable.newLevel();
    switch(p->type)
    {
    case ASSIGNMENT://赋值语句
        _assign(p);
        flag=0;
        break;
    case WHILESEN:
        flag=0;
        _while(p);
        break;
    case IFSEN:
        _if(p);
        flag=0;
        break;
    case INNERVARIDEF:
        _indef(p);
        flag=0;
        break;
    case SENBLOCK:
    //gtable.newLevel();
        Recursive(p->sonNode);
    //gtable.backLevel();
        flag=0;
```

```
        break;
    default:
        break;
    }

    if(p->sonNode&&flag)
        Recursive(p->sonNode);
    if(p->sibleNode)
        Recursive(p->sibleNode);
    //退出该层前 back
    return 0;
    }
```

六、目标代码生成

1. 读取四元式结果

根据中间代码阶段生成的四元式，初始化四元式表 formatlist，每条四元式由 op，arg1，arg2，result 以及行号组成。

```
int object::get_format()
{
    QFile format_file("format.txt");
    if(!format_file.open(QFile::ReadOnly|QFile::Text))
    {
        return -1;
    }
    QTextStream stream(&format_file);
    QString line;
    QStringList sections;
    format temp;
    int n = 1;
    while (!stream.atEnd())
    {
        line = stream.readLine(); // 不包括“\n”的一行文本
        //qDebug()<<line;
        sections= line.split(QRegExp("( |,)"));
        sections[1].remove("(");
        sections[4].remove(")");
        //for(int i=0;i<sections.size();i++)
        //    qDebug()<<sections[i];
        temp.line=sections[0].toInt();
        temp.op=sections[1];
        temp.op1=sections[2];
        temp.op2=sections[3];
```

```
            temp.op3=sections[4];
            format_list.push_back(temp);
        }
        format_file.close();
}
```

2. 初始化寄存器，符号表，标号表

根据四元式的内容得到标号表 laberlist，并且初始化 32 个寄存器为可用状态。

3. 寄存器分配和目标代码生成

- 目标代码选择 mips 汇编代码作为目标代码。
- 指令集均为 MARS 支持的指令。
- 寄存器的分配原则：尽可能留，尽可能用，及时腾空。

七、结果演示

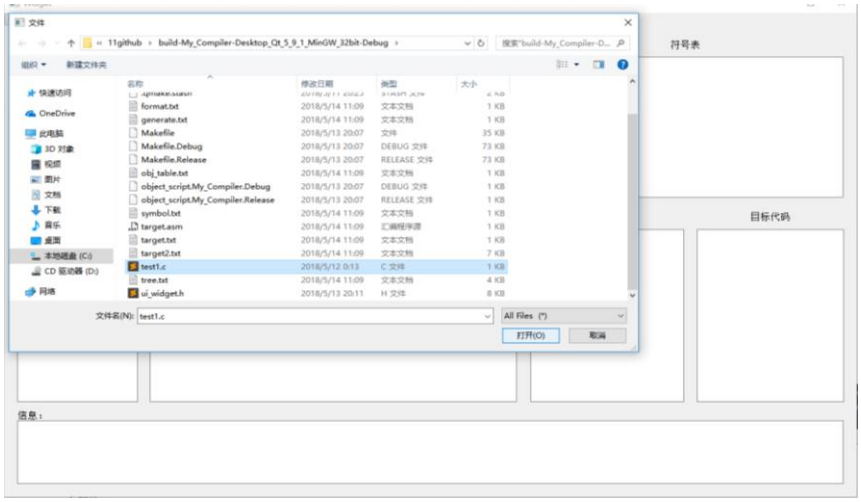


图 7-1 导入源程序

图 7-2 源程序

图 7-3 词法分析

图 7-4 语法分析

图 7-5 四元式

图 7-6 目标代码（一键完成）

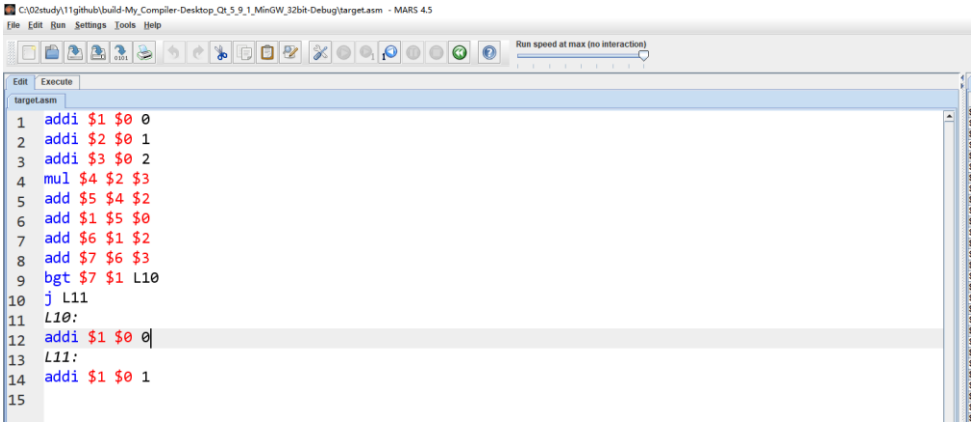
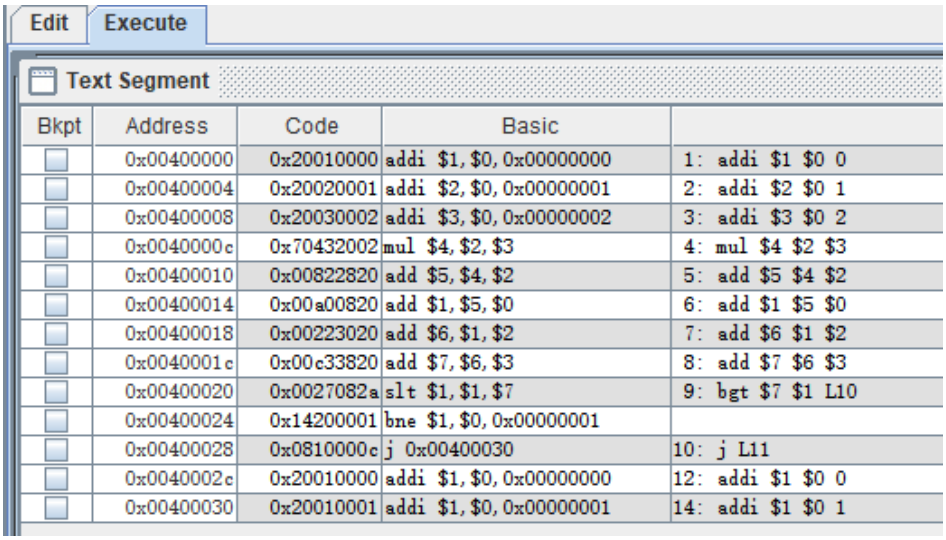
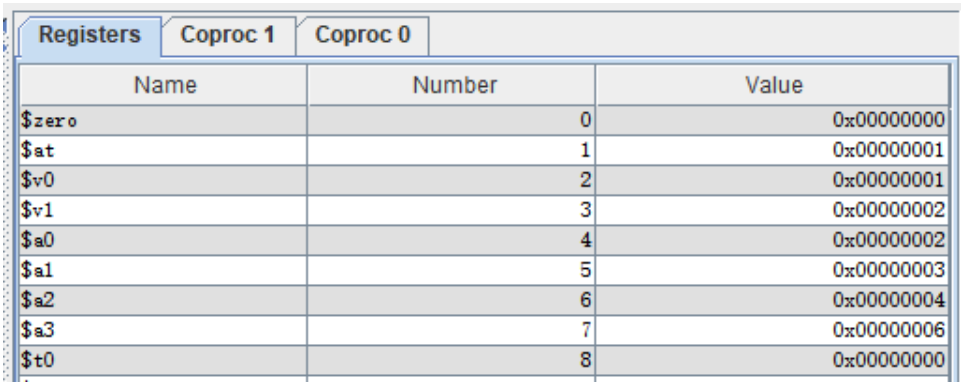


图 7-7-1 MARS 测试



Bkpt	Address	Code	Basic	
	0x00400000	0x20010000	addi $1, $0, 0x00000000	1: addi $1 $0 0
	0x00400004	0x20020001	addi $2, $0, 0x00000001	2: addi $2 $0 1
	0x00400008	0x20030002	addi $3, $0, 0x00000002	3: addi $3 $0 2
	0x0040000c	0x70432002	mul $4, $2, $3	4: mul $4 $2 $3
	0x00400010	0x00822820	add $5, $4, $2	5: add $5 $4 $2
	0x00400014	0x00a00820	add $1, $5, $0	6: add $1 $5 $0
	0x00400018	0x00223020	add $6, $1, $2	7: add $6 $1 $2
	0x0040001c	0x00c33820	add $7, $6, $3	8: add $7 $6 $3
	0x00400020	0x0027082a	slt $1, $1, $7	9: bgt $7 $1 L10
	0x00400024	0x14200001	bne $1, $0, 0x00000001	
	0x00400028	0x0810000c	j 0x00400030	10: j L11
	0x0040002c	0x20010000	addi $1, $0, 0x00000000	12: addi $1 $0 0
	0x00400030	0x20010001	addi $1, $0, 0x00000001	14: addi $1 $0 1

图 7-7-2 MARS 测试



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x00000001
$v1	3	0x00000002
$a0	4	0x00000002
$a1	5	0x00000003
$a2	6	0x00000004
$a3	7	0x00000006
$t0	8	0x00000000

图 7-7-3 MARS 测试

八、心得体会

通过一个学期的理论课程学习以及一个学期的课程设计，完成了一个比较完整的类 C 语言编译器。

首先，通过自己的设计和实现，对编译原理这门课的内容有了比较深入的了解，熟悉了变异程序的总体流程。通过编译原理的学习和实践，将计算机的一些专业课程串联起来，从操作系统，编译原理到计算机组成原理，使整个知识体系更加完整。

在程序设计方面，整个编译器的组织还是比较复杂的，编程语言采用 C++,最终结果涉及到十几个类之间的关系，因此整体框架的搭建，代码风格的同意，版本控制，兼容性，稳定性，版本控制等方面都有着比较高的要求。

在团队合作方面，由于整个项目的大部分都是由团队合作来完成，那么分工，接口的规定，时间，流程的规划都是比较复杂的问题。在此次试验中要感谢另外两位队友，在大家的通力合作下，比较顺利的完成的前期的项目。而且由于在开始之前便规定好了分工和接口，在后期个人完成以及代码重构时实现起来都比较方便。

九、参考文献

[1] Aho A V, Sethi R, Ullman J D. Compilers: principles, techniques, and tools[M]. Addison-Wesley Pub. Co, 1986.

[2] Aho A V. Compilers: Principles, Techniques, and Tools (Subscription), 2/E[M]// Compilers, principles, techniques, and tools /. Addison-Wesley Pub. Co. 1986.

[3] 《程序设计语言 编译原理（第三版）》，陈火旺，国防工业出版社

十、附录：源代码(部分)

1. 词法分析部分

```
int LexAnalyzer::startScanner()
{
    char ch = fgetc(source);
    int states = 0;
    while (!feof(source) && states == 0)
    {
        if (ch == ' ' || ch == '\t' || ch == '\r' || ch == '\l')
        {
            ;//nothing jump it!
        }
        else if (ch == '\n') {
            addNewLine();
            clearState();
        }
        else if (ch == '/')
        {
            comment(ch, 0);
            clearState(); //清除状态信息
        }
        else if (ch >= '0'&&ch <= '9')
        {
            states = number(ch, 1); //处理数字
            clearState();
        }
        else if (isOperator(ch))
        {
            states = myOperator(ch, 2); //处理操作符
            clearState();
        }
        else if (isLiter(ch))
        {
            states = identifier(ch, 3); //处理标志符
            clearState();
        }
        else
        {
            return lineNum;
        }
        ch = fgetc(source);
    }
```

```
    return states;
}
```

2. 语法分析部分

```
int grammer::analyze(string filename)
{
    if (grammerTree != NULL) {
        delTree(grammerTree);
        grammerTree = NULL;
    }
    vector<int> levelCount = vector<int>(128, 0);
    vector<int> level = vector<int>(1024, -1);
    int level2[1024];
    treeNode **stack2 = new treeNode*[1024];
    ifstream fin(filename, std::ios::in);
    if (!fin) {
        //cout << "OPEN SOURCE FILE ERROR" << endl;
        return -1;
    }
    ofstream fout(OFileName, std::ios::out);
    ofstream fout2("tree.txt", std::ios::out);
    if (!fout) {
        return -1;
    }
    string line;
    int lineNum = 1;

    int point = 0;
        //始终指向栈顶元素
    int point2 = 0;
    localStack[point] = END;
    grammerTree = new treeNode();
    grammerTree->type = PROGRAM;

    localStack[++point] = PROGRAM;
    stack2[point] = grammerTree;
    level2[0] = 0;
    level[point] = 0;
    bool flag = true;
    treeNode *nodeTemp;
    getline(fin, line);
    string v;
    int attribute;
    SplitString(line, v, attribute);
```

```
    int count = getToken(v);
    while (flag) {
        while (count == NL) {                                    //忽略换
行符
            lineNum++;
            getline(fin, line);
            SplitString(line, v, attribute);
            count = getToken(v);
        }
        if (localStack[point] < tokenNum) {
            if (localStack[point] == count) {
                for (int i = 1; i < level[point]; i++)
                    if (levelCount[i] != 0) {
                        fout << special[0];
                        fout2 << " ";
                    }
                    else {
                        fout << special[2];
                        fout2 << " ";
                    }
                    fout << special[1];
                    fout2 << " ";
                    levelCount[level[point]]--;
                    fout << strTokens[localStack[point]];
                    fout2 << localStack[point];

                    switch (localStack[point])
                    {
                    case ID:
                    case NUM:
                    case OP1:
                    case OP2:
                    case RELOP:
                        fout << " - " << attribute << ' ';
                        fout2 << " - " << attribute << ' ';
                        break;
                    default:
                        break;
                    }
                    fout << endl;
                    fout2 << " " << lineNum << endl;
                    if (getline(fin, line)) {
//读取下一个字符，若已经读至文件末尾，返回-1，证明语法正确，检验成功
                        SplitString(line, v, attribute);
```

```
                    //cout << v[1] << endl;
                    count = getToken(v);
                    point--;
                    continue;
                }
                else {
                    fin.close();
                    fout.close();
                    fout2.close();
                    generateTree();
                    for (int i = 0; i < 20; i++)
                        varTable.newTemp(i);
//                    outfss.open("txte.txt");
//                    treeTra(grammerTree, 0);
//                    outfss.close();
                    cout << localId << endl;
                    return 0; //正确判断的输出结果为0
                }
            }
            else {
                flag = false;
            }
        }
        else if (localStack[point] == END) {
            if (localStack[point] == count) {
                levelCount[level[point]]--;
                point--;
                continue;
            }
            else {
                flag = false;
            }
        }
        else if (localStack[point] >= PROGRAM && localStack[point] <=
FACTOR) {//栈顶为非终结符
            for (int i = 1; i < level[point]; i++)
                if (levelCount[i] != 0) {
                    fout << special[0];
                    fout2 << " ";
                }
                else {
                    fout << special[2];
                    fout2 << " ";
                }
```

```
if (localStack[point] != PROGRAM) {
    fout << special[1];
    fout2 << " ";
}

fout << strTokens[localStack[point]];
fout2 << localStack[point];
switch (localStack[point])
{
case ID:
case NUM:
case OP1:
case OP2:
case RELOP:
    fout << " - " << attribute << ' ';
    fout2 << " - " << attribute << ' ';
default:
    break;
}

sentence *s = new sentence;
s = matrix[localStack[point] - tokenNum][count];
if (s->tokens[0] == ERROR) {
 //为空，则发现语法错误，调用出错处理程序进行处理
    flag = false;
}
else if (s->tokens[0] == EPSILON) {
 //A→ε ，则只将 A 自栈顶弹出。

    fout << " - @";
    fout2 << " - @";
    levelCount[level[point]]--;
    point--;
}
else {
    int j = level[point];
    levelCount[j]--;
    /*                    nodeTemp = stack2[point];*/
    int ktemp = point;
    for (int i = s->tokenLen - 1; i >= 0; i--) {
        localStack[point] = s->tokens[i];
        level[point] = j + 1;
        levelCount[j + 1]++;
        point++;
```

```
                    }
                    point--;

                    /*
                    treeNode *sonNode3, *sonNode2;
                    sonNode3 = sonNode2 = new treeNode;

                    for (int k = point; k > ktemp; --k) {
                    stack2[k] = sonNode3;
                    sonNode3->type = stack[k];
                    sonNode3->sibleNode = new treeNode;
                    sonNode3 = sonNode3->sibleNode;
                    }
                    sonNode3->type = stack[ktemp];
                    nodeTemp->sonNode = sonNode2;*/
                }
                fout << endl;
                fout2 << " " << lineNum << endl;
            }
        }
        //cout<<"出错行位置"<<line;
        return lineNum;
}
```

3. 中间代码部分

```
/*
1. 声明语句。 newTemp， 如果返回错误，输出，x 行重复定义。
2. assign 语句， 左边的 id 必须在表达式计算完以后，调用 dirty 函数。
3. sentence block 进入 son 节点时候，调用 newLevel， 离开调用 backLevel。
*/
int Generate::Recursive(treeNode *p)
{
    //qDebug()<<strTokens[p->type]<<"   "<<p->value<<"   "<<p->lineNum;
    int flag=1;
    //进入语句块
   // gtable.newLevel();
    switch(p->type)
    {
    case ASSIGNMENT://赋值语句
        _assign(p);
        flag=0;
        break;
    case WHILESEN:
```

```
        flag=0;
        _while(p);
        break;
    case IFSEN:
        _if(p);
        flag=0;
        break;
    case INNERVARIDEF:
        _indef(p);
        flag=0;
        break;
    case SENBLOCK:
    //gtable.newLevel();
        Recursive(p->sonNode);
    //gtable.backLevel();
        flag=0;
        break;
    default:
        break;
    }

    if(p->sonNode&&flag)
        Recursive(p->sonNode);
    if(p->sibleNode)
        Recursive(p->sibleNode);
    //退出该层前 back
    return 0;
    }
```

4. 四元式部分

```
void detectLabel(vector<string>& v, int addr)
{
    int i;
    if(v[0][0]=='L'){
        for(i=0;i<=cnt;i++){
            if(v[0]==labellist[i].Lname)
                break;
        }
        if(i>cnt){
            labellist[cnt].Lname=v[0];
            labellist[cnt].Laddr=addr;
            cnt++;
        }
```

```
    }
}


//生成四元式
void fourformat(vector<string>& v, int addr, string &code)
{
    int i, jaddr;
    if(v[0][0]=='L')  //去掉标号带来的影响
        v.erase(v.begin());
    stringstream ss;

    if(v[0]=="goto"){
     //goto 语句，直接 jump，生成 addr (j,-,-,jaddr)的格式
        string temp=v[1]+":";
        for(i=0;i<=cnt;i++){
            if(temp==labellist[i].Lname){
                jaddr = labellist[i].Laddr;
                //cout << addr << " (j,-,-," << jaddr <<")" << endl;
                ss << addr << " (j,-,-," << jaddr <<")" << endl;
                code=ss.str();
            }
        }
    }
    else if(v[0]=="if"){
//if 语句，条件转移，生成 addr (jrelop,x,y,jaddr)，relop 为< > <= >= == !=
        string temp=v[v.size()-1]+":";
        for(i=0;i<=cnt;i++){
            if(temp==labellist[i].Lname){
                jaddr = labellist[i].Laddr;
                ss << addr << " (j" << v[v.size()-4] << "," <<
                       v[v.size()-5] << "," << v[v.size()-3] << ","
                       << jaddr << ")" << endl;
                code=ss.str();
            }
        }
    }
    else if(v[1]==":="){ //赋值语句
        if(v.size()==3){ //直接赋值，三个参数
            ss << addr << " (" << v[1] << "," << v[2] << ",-," <<
                   v[0] << ")" << endl;
            code=ss.str();
        }
        else{ //计算赋值，四个参数
```

```
                ss << addr << " (" << v[3] << "," << v[2] << "," << v[4]
                            << "," << v[0] << ")"<< endl;
                code=ss.str();
            }
        }
    }
```

5. 目标代码部分

```
int object::object_code()
{
    QFile target("target.asm");
    if(!target.open(QFile::WriteOnly|QFile::Text))
          qDebug()<<"error";
    QTextStream out_t(&target);


    format temp;
    //format_list.size();
    //qDebug()<<format_list.size();
    for(int i=0;i<format_list.size();i++)
    {

        temp=format_list[i];
        for(int i=0;i<laberlist.size();i++)
        {
            if(temp.line==laberlist[i])
                out_t<<"L"<<temp.line<<":"<<endl;
        }
        //qDebug()<<temp.op<<endl;
        if(temp.op.toStdString()==":=")
        {
            if(temp.op1[0]>='0'&&temp.op1[0]<='9')
            {
                int reg_num=is_alloc(temp.op3);
                if(reg_num<0)//该变量未在寄存器当中,那么就加进去
                {
                    reg new_reg;
                    new_reg.name_id=namelist.indexOf(temp.op3);
                    new_reg.value=temp.op1.toInt();
                    //cout<<new_reg.name_id<<" "<<new_reg.value<<endl;
                    if(next_reg>=31)
                        next_reg=1;
                    reg_list[next_reg]=new_reg;
```

```
                    out_t<<"addi $"<<next_reg++<<" $0
"<<temp.op1.toStdString().c_str()<<endl;
                }
                else//已经在寄存器当中
                {
                    out_t<<"addi $"<<reg_num<<" $0
"<<temp.op1.toStdString().c_str()<<endl;
                }

            }
            else//寄存器之间的赋值
            {
                int reg_s=is_alloc(temp.op1);
                int reg_d=is_alloc(temp.op3);
                out_t<<"add $"<<reg_d<<" $"<<reg_s<<" $0"<<endl;
            }

        }
        else if (temp.op.toStdString()=="+")
        {
            int reg_num=is_alloc(temp.op3);
            int reg_s1=is_alloc(temp.op1);
            int reg_s2=is_alloc(temp.op2);
            if(reg_num<0)//该变量未在寄存器当中,那么就加进去
            {
                reg new_reg;
                new_reg.name_id=namelist.indexOf(temp.op3);
                new_reg.value=temp.op1.toInt();
                //cout<<new_reg.name_id<<" "<<new_reg.value<<endl;
                if(next_reg>=31)
                    next_reg=1;
                reg_list[next_reg]=new_reg;
                out_t<<"add $"<<next_reg++<<" $"<<reg_s1<<"
                      "<<reg_s2<<endl;
            }
            else
            {
                out_t<<"add $"<<reg_num<<" $"<<reg_s1<<"
                    $"<<reg_s2<<endl;
            }
        }

        else if (temp.op[0]=='j')//跳转类型
        {
```

```
        laberlist.push_back(temp.op3.toInt());
        if(temp.op=="j")
        {
            out_t<<"j L"<<temp.op3.toStdString().c_str()<<endl;
        }
        else if(temp.op=="j=")
        {
            int reg_s1=is_alloc(temp.op1);
            int reg_s2=is_alloc(temp.op2);
            out_t<<"beq $"<<reg_s1<<" $"<<reg_s2<<"
                L"<<temp.op3.toStdString().c_str()<<endl;
        }


    }
}
target.close();
return 0;
}
```